DØ Global Tracking Review

Jim Kowalkowski
Marc Paterno

1 Introduction

This review covered a very large collection of packages, falling into the following groups:

1. **trf++** utility packages. These provide basic utilities used by **trf++**, but which are not directly related to tracking; for example, **ptr** and **trfutil**.
2. **trf++** base packages. These packages form the **trf++** framework.
3. Detector specific packages. These classes use the tools in the **trf++** base packages to perform tracking in each subdetector. We have concentrated on the SMT detector's software.
4. Global tracking packages. These classes drive the tracking process from the highest level, and provide the DØ framework *Package* subclass that forms the basic tracking package for DØReco.

This set of packages is significantly larger than that we have considered in any previous review. Therefore, we have not been able to review all the material to the depth that we have done in previous reviews. In response to the guidance provided by management, we have concentrated on issues of efficiency — meaning the speed of the reconstruction code, not the tracking efficiency. The best method for determining the speed of the code, and for finding the parts of the code that are most in need of speed optimization, is to profile an optimized executable. We were unable to obtain an optimized executable in the time available for this review. We recommend that a follow-up study of an optimized executable be undertaken as soon as possible. We would be happy to be part of such a study. In the absence of such a study, this document provides a starting point for discussion of how to improve the tracking code.

In the absence of an optimized executable, we studied the structure of the code, using the SMT tracking code as an example. We concentrated on the use of the **trf++** tools, and studied the implementation of the **trf++** toolkit only as was necessary to understand the higher-level code using those tools.

It is important to note that many of the observations and conclusions within this document come from code sampling. We pulled out samples of code for most of the layers and processing and went through it in detail. The information contained in this document refers to code that lives mostly in the SMT and the **trf++** packages.

2 Overview

The documentation by David Adams, and made available on the web, provides a good general overview of the **trf++**, and to some extent, DØ's use of **trf++**. The **trf++** framework is quite general, and provides a full set of facilities for building tracking software. In addition to the classes directly supporting the task of finding tracks, **trf++** provides the following:

1. a persistent object model, used primarily to save the state of the "algorithm" objects or construct objects in a particular state that are used in the system;
2. matrix and vector classes, used in the representation of positions, track parameters, and the uncertainties associated with these quantities;
3. "smart pointer" class templates, providing a variety of memory management schemes and the ability to convert an object into a index form to be stored by EDM/dØom or referenced later;
4. a simple event model, used to store the "clusters" and tag events with an ID used in track finding;

5. a set of header files defining constants;

6. a simple geometry package.

We found nothing in the basic design of **trf++** that would indicate it would be inherently inefficient, and therefore see no need for a significant redesign. All our suggestions are in the realm of implementation, where additional attention to detail could enhance the maintainability and speed of the code, with little or no loss of flexibility.

3 Major Concerns

In general, we have found many of the same things here that were found in our review of the muon reconstruction code. The muon review document is available on our web site[1], and the material will not be duplicated here. The next major section will, however, include examples of where we have found these problems when appropriate. The sections of the muon review document which have relevance here are:

- *Error reporting* (Section 3.1.2). It is unclear how error (exceptions/return codes) are handled and reported in this system.
- *Problem Decomposition* (Section 3.2.2). We looked through several methods of classes that were in excess of 200 lines. This makes us immediately wonder if the problem has been decomposed correctly. The lack of object-oriented nature makes such code harder to understand, and thus less likely to be efficient or correct. Code that is difficult to read is also difficult to maintain, which is an especially painful burden in the algorithm code, where most of the maintenance and improvement effort is likely to be concentrated.
- *Common Tools and Libraries* (Section 4.1). The tracking system contains a large number of utility classes and code that also exists elsewhere in the DØ infrastructure.
- *Classes with Useful Features* (Section 4.2).The concept of data objects goes too far in many places. Classes that use the data objects pull information out and do common manipulation in many different places; this report goes into more detail about this sort of problem.
- *Example MuoHitProcessor Changes* (Section 4.3).This example shows a detailed explanation of what can be does to improve a class and make the code that uses it much more maintainable.

We refer the reader to the muon software review document for more detail on these observations.

As the major goal for this review was to look into the speed of execution of the global tracking software, we concentrated on issues specifically related to the speed of the code. A few major problems popped out as we went through the code. Some of these are known to cause performance problems while others are subject to debate, and should be the subject of further investigation using the profiler.

It is important to note that we have concentrated not on the individual routines which consume the greatest amount of time (attempting to identify them without the profiling of an optimized executable would be mere guesswork), but rather on the sorts of smaller (but pervasive) inefficiencies that can "nickel-and-dime" a program to death.

3.1 Excessive Copying

There are many places in the code where large or complex items are passed by value; often these are collections which are created in a function, and returned (by value) as the return value of the function. STL containers are not cheap to copy and return by value. The muon review document contains suggestions and alternatives to this approach. In many places we see data being copied out of objects, just so that data may be used locally, for the duration of one function. This (usually needless) copying was present in so many places that it caught our attention as a potential performance problem.

There are many parts of the design where items (such as clusters) are copied from one place (for example, the chunks in the EDM) to another place (the registry which seems to be the event model of **trf++**). Is this

[1] The URL for our muon reconstruction software review document is:
http://cdspecialproj.fnal.gov/d0/muonreview/index.html.

done because EDM is not adequate? Is it does so that **trf++** /GTR packages can communicate outside the framework communications channels? If the EDM is inadequate for the needs of global tracking, it should be modified. If this is done to retain “experiment independence” of **trf++**, then DØ should evaluate the costs and benefits of this feature. A meaningful measurement of the cost is probably not possible without the results of profiling an optimized executable.

Object translations are a form of copying and are sometimes necessary. In the case of objects that exist in large quantities and consume a large amount of resources such as memory, they should be avoided. Consider the SMT code. Here dØom reads in the arrays of data (DSPACK/EVPACK) that represent objects. It then translates the data to C++ objects. The C++ objects are then translated into **trf++** objects. Manipulations such as this are costly and layers of translation should not exist without a good reason; again, DØ must weigh the cost in efficiency against the benefits.

3.2 Poorly Organized Objects

Poorly organized objects contain data members that do not properly describe the things the object owns and manipulates. They are often bloated and can lead to performance and maintenance problems.

Many classes in this system have data members that refer to other data through pointers using inappropriate data structures. This is typically done through use of STL containers such as *map*, *vector*, or *list*, or the **trf++** class template *array*. Many times the quantities that these containers hold are very specific things such as 2x2 matrices or a point in space and can be directly represented by specific class. A poorly organized object will end up with a lot of overhead that will actually cause confusion for a person that is maintaining the system.

An example is the measurement with coordinates (*v*, *z*). This point can be represented as an object with two doubles named iv and iz, referred to as point.v( ) and point.z( ). Manipulating this object (copying it, or merely accessing the members) is efficient in terms of space and time and easy to grasp. In addition, we can define operations on this point object.

This point can also be represented by an *array<double>* that is initialized to have two elements. Now we pay an additional price to copy the point, because of the generality of *array* (we have to pay for the extra pointer and an extra integer, to keep track of the size of the array) and we must do a pointer dereference to access the members. We also must refer to the values in the array using enumerated values such as IV=0, IZ=1, using *point[IV]* and *point[IZ]* to access the data. Defining specific operations (related to its being a two-dimensional point) on this object does not make sense because it is just an array; therefore the data must be manipulated outside the object itself. The only clue that one gets of the meaning of this object is through the comments in the code. It is more difficult to grasp the concepts for manipulating this object.

Another example is the class *SpacePoint*. For this class, we pay the price of the virtual table pointer when most, if not all of the uses to not need something like this. Furthermore, we pay the price of moving seven doubles (plus the vtable pointer) around instead of just three doubles. The *SpacePoint* class defines no arithmetic operations such as scale, distance from, or addition. Without these useful features, developers end up pulling the values out and manipulating them outside the object. This is a maintenance problem as well as a source of bugs; it also causes many different solutions to be coded for the same problem.

3.3 STL Container Usage

This document goes into detail about specific issues regarding STL container use in later sections. We believe that using the STL containers is a very good thing when the problem calls for them. We also believe that care must be taken to ensure that the application will perform well when STL containers are used. A list of a few observations follows.

- We found widespread use of *map*s and *multimap*s. For some problems, such as configuration management, these are the best solutions. Sometimes they are overkill or not used properly. It is important to note that *map*s are complex structures for which it costs time to insert and erase items.
- We found widespread use of *vector*s in places where specific specialized objects may be more appropriate. When *vector*s are used, giving an initial size estimate using reserve that greatly improve performance.

- We found widespread use of *list*s, especially as return-by-value objects. Lists are not cheap to construct or copy, they can involve many heap memory manipulations. Often there was no obvious reason for the use of *list* rather than *vector*; the problem at hand had no need for the manipulations at which *list* excels. In the absence of such need, *vector* is the better choice.

3.4 Objects Without Proper Behavior

This is a general problem that we see in many different pieces of software. Objects do not contain code that is useful in manipulating them, so each user pulls out the appropriate values and does the calculation. There is a difference here between a specialized algorithm that must manipulate the data of an object directly, and common manipulations that belong in the object itself. A simple example is a creation of a unit matrix. The user should not need to access each element of the matrix and set it to zero or one, it should be able to call a method that initializes the matrix to a unit matrix. Even specific algorithms can be encapsulated in an adapter class that is constructed using a reference or pointer to an instance of the data class. This type of organization is described in the muon review document in the hit manipulation code section. We believe that this type of organization greatly improves the understanding and maintainability of the code. It also improves the robustness of the code.

3.5 Old Code

We see much use of *array* and *matrix* class templates. Each of these utilities class appear to have counterparts in one or more of the STL, or the Zoom or CLHEP libraries, as well as other packages such as the Standard Matrix Library (a library that we are looking into)[2]. Why do these class templates exist and what is their advantage? They do not appear on the surface to provide any benefit and seem to be an extra maintenance burden, as well as a potential performance problem. These classes also appear to be overused to represent concepts that are really quite different then arrays and matrices — such as the point or measurement discussed in an earlier section.

3.6 Memory Management

We are concerned that the **ptr** package is too much a heavy-weight management scheme. In his popular book **Effective C++**, Scott Meyes illustrates a technique that seems like it may be more appropriate here. Basically, each object can contain an integer that is used for reference counting and the *Ptr* template would manipulate that reference count. This simple change may significantly effect performance. Without profiling, it is difficult to be sure of this; however, there are commonly-agreed-upon the reasons we expect this to be true. This is discussed in more detail in Section 6.1, on page 11.

Let us briefly exam the effects of *Ptr* on short-lived objects such as clusters. By shorter-lived, we mean that many are constructed and destroyed for each event. For each one of these objects, we must construct a companion reference count object on the heap; its lifetime is the same as the object that it is paired with. Depending on the object size, this overhead in terms of time and space can be significant. The *Ptr* class itself appears to consist of three pointers (one pointing to the shared management policy table, one pointing to the object itself, and one for the vtable of the *Ptr<T,P>* class), so copying a *ClusterList* will involve the copy of three pointers for each element in the list. Copies of *ClusterList* objects are made all over the system. A simpler smart pointer system would only involve having one pointer, reducing the size by a factor of three. An additional benefit would be that the reference count would be stored close by the data of the object, not in a separately allocated block of data. This locality of reference could yield better memory cache performance.

The conclusion here is that smart pointers are very important in this system. We believe that the current implementation is not the best choice for this type of problem and can be improved.

[2] The Matrix Template Library (MTL) and documentation for the library is available on the web, at URL http://www.lsc.nd.edu/research/mtl.

4 Coding Recommendations

We have divided the subjects in this section into two main parts: issues of *maintainability and flexibility*, and issues of *efficiency*. Of course, some issues affect both maintainability and efficiency; in those cases, we have filed the issue under what seems to us to be its most important facet.

4.1 Maintainability

4.1.1 Naming Conventions

The names of parameters to methods and function did not generally give any indication as to what the quality or object represents. Names for parameters and variables should indicate roles or something useful and not just an abbreviated form of the class name.

4.1.2 Obs files and Object Persistence

The *ob*s files contain the information that controls the behavior of the fitting program, by specifying the paths that are followed in the tracking process. It does not appear that this information is preserved in the reconstructed data; there is no equivalent of the RCP database or the functionality of *RCPID*. The Offline Reconstruction group must decide if this is an important feature for the global tracking system to have. If so, then it must be decided whether to add these features to the **objstream** package or to have the algorithms initialized using *RCP* objects.

There appears to be a need for a certain set of parameters (*e.g.* minimum transverse momentum, maximum distance of closest approach) that are of global significance for the tracking; the full set should be identified by the experts, and they should be pulled out and put into one place for easier modification. Currently these parameters are scattered throughout many *obs* files and require an expert to change. Since the parameters are scattered, it is difficult to be sure that all are covered when a change is made.

We understand the *obs* files and **objstream** package to be a light-weight persist object package and that this package exists because RCP and dØom were either not adequate or not experiment independent. This should be reexamined. DØ standardized on dØom and it appears that much effort was expended to produce a second persistency mechanism here in the tracking code. The tracking code is huge, why add all this utility code bulk to the maintenance effort? An experiment independent package can make the assumption that the experiment has a persistency package and build an abstract interface that allows coupling to it; the maintenance burden of such an interface would be considerably less than that of the full persistency mechanism.

4.1.3 Constants

In **trfutils** there is a definition of constants PI, HALFPI, etc. These all suffer from the "order of static initialization" problem — if another translation unit makes use of these quantities in its own static initialization, the language does not guarantee in what order the initialization is to be done. Such constants should be replaced by static functions, which can return the constants. These functions can be declared *inline*, and in an optimized executable will incur no function call overhead. We know of several methods of producing real constants that are inlined (no symbol table or program bss/data section entries) as opposed to data member in the executable. The optimizer can do very good things with real constants.

4.1.4 Namespace pollution

Many of the headers pollute the global namespace by hosting some classes from other namespaces (often but not only *std*) into the global namespace. At least six headers in five different packages have *using namespace std* in a header; more than 130 headers have at least one *using std::**c***, for some class ***c*** defined in the *std* namespace. Some of these headers are elements of lowest-level tools (such as *Ptr.h*, in the package **ptr**) which are included in many other files, so the problem is endemic.

4.1.5 Insufficient error handling

There are many places in the code where we found insufficient error handling. Generally, this is of the form "if something goes wrong, write to *cout*, and return a status code". Unless the calling code always checks the status code (it does not, and there is no way to have the compiler require this), the result is an error message lost in the blizzard of output produced by DØReco, and an error waiting to happen. For example, in **gtrprop**, the class *SMTPropagator* prints messages to *cout*, and then returns a failure. The messages to *cout* is not appropriate.

It would be far better for errors to be indicated by an exception throw; this allows information to be passed up to a higher level of processing, where an intelligent decision can be made on how to proceed.

Exceptions thrown as shown below will not be at all useful. There is nothing that is set up to catch this type of exception and the message will not be displayed at all, instead a message "Unknown exception caught" will be displayed.

```
if (_reco->this_alg==0) throw "No acceptable algorithm Chosen.";
```

A slightly better way would be to throw *runtime_error("No acce…")*. A truly useful message would also contain information identifying the place at which the trouble was detected, and whatever data caused the failure.

4.1.6 Multiple Implementations of One Concept

There seem to be quite a few classes adding to the bulk of the code base, requiring extra maintenance, but giving little payoff, because these classes are reproducing what already exists in the standard library. In some cases, it seems that these classes may be a legacy from early C++ library implementations. Such code should be modernized, to take advantage of the Standard Library where available. In other cases, packages for the different subdetectors produce their own classes representing the same abstraction. This creates additional bulk. It also ensures that at least one implementation of the given abstraction is non-optimal.

Some examples follow:

1. In **trfzp**, *SurfPolygon.h* introduces its own class *Pair*. It looks quite like *pair<double, double>*.
2. *nvector* is very similar to *valarray*.
3. **trfutil** provides the classes *matrix* and *smatrix*. Some of the SMT packages use *vector<vector<double> >* to represent a matrix, rather than using *matrix* or *smatrix*. What are the benefits of having *matrix* and *smatrix*, rather than using an established linear algebra library?

4.1.7 Use of private code

A number of compilation units include files from the **framework/private** directory — which the documentation (not to mention the naming of the directory itself) says should never be done. The interfaces in these files are not published and are subject to change at any time. Because C++ given no language mechanism by which such illegal usage can be prevented, we must rely on the software designers themselves to not violate such clear design features.

4.1.8 Missing inclusion guards

A handful of headers do not include preprocessor guards against multiple inclusion. This is clearly dangerous. While these headers may currently be used in a manner that does not cause trouble, this is a maintenance accident waiting to happen.

The list of headers which have no include guards are:

- *gtr_evt/src/GTrackChunkTest.hpp*
- *gtr_evt/src/McTrackChunkTest.hpp*
- *objstream/src/ObjData_inlined.hpp*
- *objstream/src/ObjStreamTest.hpp*

- *objstream/src/ObjTableTest.hpp*
- *objstream/src/ObjTable_inlined.hpp*
- *objstream/src/ObjTypeTest.hpp*
- *smttrack_reco/src/TrackAnalyzer.hpp*

4.1.9 Incorrect use of *auto_ptr*

The class *SMTGlblBCollectChunk* makes use of the fact that KCC 3.3 has an incorrect implementation of *auto_ptr*. The problem is that (according to the Standard) the copy constructor for *auto_ptr* takes a non-const reference to another *auto_ptr* (not a const reference); when a copy is made, the old *auto_ptr* loses ownership of (and access to) the object which is pointed to. A library that has a standard-conforming implementation (as does KCC 3.4) will cause a failure when an instance of this class is copied. We should note that the library that ships with MSVC++ also has the implementation of *auto_ptr* done incorrectly; at least only other library is freely available and does not have this flaw[3].

In general, it is important to review all use of *auto_ptr* throughout the code, because of the change in behavior from that mandated by the earlier drafts of the Standard to that mandated by the (final) Standard.

4.1.10 Noisy output

GtrFind::process_event( ) writes to *cout* on every event. A policy should be set up in the framework for allowing packages to produce output that can be filtered. If the output is an error or warning message, then the Zoom error logger must be used.

4.1.11 An obscure idiom

In the file *SMTLocalGlobalTrans.cpp* (from **smt_hitalgs**) we find the following code snippet:

```
if(! _data->first) d1=1,d2=2;
```

This makes use of the C++ comma operator (which is rarely used, and therefore is a source of maintenance mistakes). It uses the fact that the two expressions separated by the comma operator form a single statement, and thus do not need to be enclosed in braces. This is very dangerous, since someone less knowledgeable about C++ than the code's original author could easily introduce a subtle mistake that would not be a compilation error, but would lead to incorrect runtime behavior.

4.2 Efficiency

4.2.1 Inappropriate generality

Several classes in the tracking system suffer needless overhead because of classes which are overly general — that is, they have flexibility that is not need, but still must be paid for, in either time or space, or both.

4.2.1.1 Inappropriate use of dynamically sized collections

The class templates *vector* and *array* (from **trfutil**) are both dynamically-sized collections. This comes at a cost: the collected objects must be stored on the heap (thus degrading locality of reference, when many such collections are used together), and extra memory is required, to hold the pointer to the data and the number of elements stored — this overhead can be significant if the collection itself is small[4]. The poor locality of reference may be more costly than the extra bulk of the objects, since it will increase the frequency of cache misses — and modern processors slow dramatically whenever a cache miss occurs.

[3] The library in question is STLport; it can be downloaded from, and documentation found on, the web at URL: http://www.stlport.org.

[4] The class template *array* suffers from the additional overhead of a vtable pointer.

For example, *SMTHit* (from package **smt_hit**) uses *vector<float>*, fixed to a size of 5, to represent a combination of 2D and 3D positions. There is no need for the power of *vector* here, since there will never be more than 5 elements. Furthermore, it is not clear why there isn't a class — or possibly two classes — to represent this sort of position. It seems that this class consists of two different parts, and that the two parts are used by different clients. An even worse example is a matrix represented as *vector*< *vector*<double> >, which is found in *SMTXyzGlbl* from package **smt_hit**; this is not the only place where this can be found.

4.2.1.2 Inappropriate inheritance

Some classes inherit from base classes that provide functionality that is not needed by the derived class. When this functionality comes with an associated cost, one must consider if the associated cost is sufficiently large to make the inheritance inappropriate.

The class templates *nvector*, *matrix* and *smatrix* (all in **trfutil**) all inherit from the class template *array*. As mentioned in Section 4.2.1.1, *array* is a dynamically-sized object. Many of the uses of *nvector*, *matrix* and *smatrix* have no need for this feature, and thus should not pay the cost for having it. The cost are particularly high in the most commonly used cases of 2×2, 3×3, … 6×6 matrices. In these special cases, generality is not useful, but it is costly.

The classes *ClusZPlane1*, *ClusZPlane2*, *ClusZYPlane1*, *ClusXYPlane2*, *ClusCylPhi*, and *ClusCylPhiZ* all inherit from the class *McCluster*, which represents a simulated cluster. The base class holds a *McIdList*, which is a typedef for *vector<int>*. Thus every instance of each of these classes, even when created from collider data, must pay the price for carrying an empty *vector<int>* — which , for KAI C++ on a 32-bit system, is 16 bytes per cluster (a *vector* holds an allocator and three pointers). For an event with tens of thousands of clusters, this is a considerable amount of memory, which will further degrade the locality of reference of the code using it.

4.2.2 Useless copying

There are a few places where we found apparently needless copying. This may be a result of code maintenance — that is, code that has developed over time. When the code is not easy to read, maintenance can easily lead to leftover sections of code which are no longer performing a useful function. It is often hard to identify such portions of code. For example, in *GtrFind.cpp* (package **gtr_find**), there is a loop that copies a list of *ChunkIDs* into another list, inside a loop, in an slightly obscure idiom[5]:

```
GTrackChunk::ChunkList parent_chunks;
parent_chunks.insert( parent_chunks.end(),
                      cluster_chunks.begin(), cluster_chunks.end() );
```

Neither the original list nor the copy is ever modified. Both are passed (by reference) into a constructor of *GTrackChunk*. This would seem to be an example of needless copying, but the size of this piece of code (this one function, *GtfFind::process_event( )* is nearly 200 lines long) makes it hard to tell.

Below is a section of inefficient code. The method "*measured_vector*" returns a *HitVector* by value. The code that uses it makes two temporary copies of the *HitVector* only to pull out a single value. The work of creating and destroying the temporary copies of the actual *HitVector* is a waste of time. This pattern or style of coding is found in many places.

[5] It is unclear why the copy constructor of *ChunkList* is not used here, since *cluster_chunks* is also a *ChunkList*.

```
// return the measured hit vector
virtual HitVector measured_vector() const =0;

// Fetch u,v,z and plane phi1 for the first hit.
const Surface& srf1 = phit10->get_surface();
double phi1 = srf1.get_parameter(iphi);
double u0   = srf1.get_parameter(idist);
double v0   = phit10->measured_vector()(0);
double z1   = phit10->measured_vector()(1);
```

4.2.3 Copying of SMT cluster data

The SMT cluster data is copied from its original “chunk” format to its **trf++**-based format. This involves a significant amount of work, for little if any gain. These data should be handled in the same manner as the cluster data from the CFT, which is created directly in a format usable by **trf++**.

5 Design Issues

5.1 Framework Use

Organization, control, and running of the tracking framework executables may be improved by using the framework’s grouping feature. The descriptions of the SMT packages can be organized into a single RCP file that can be included at a higher level. When the SMT group RCP file is included, all the packages within it will be run in the proper order and the proper time. Other parts of the system such as the CFT may also benefit from this organization.

The standalone object files can be used to cause multiple packages to be registered with the framework factory. This may simplify creation of executables by introducing an entire set of package that perform a function all at once, without such a proliferation of standalone object files.

Most of the framework packages still contain all the old-style interface requirements. Nothing needs to be provided in a subclass of a framework *Package* except a constructor. This means that the *packageName( )*, *package_name( )*, and *version( )* methods are never used. *Package*s should provide a method *statusReport( )*.

The following registration with the framework is not appropriate, it results in a string like “SMTClusterPack $Id” to appear in the framework registration table, which is not at all useful. A simple way to do this is correctly is to get rid of the *version( )* method (it is not necessary) and put “$Name: ” (note the space) directly into the second parameter of the macro. The variable $Id should be avoided; it is the RCS ID of the current file and is not very useful, even if the substitution occurred correctly. The variable $Name is the tag for the current CVS package. This is what most of the other packages use.

```
FWK_REGISTRY_IMPL(SMTClusterPack,SMTClusterPack::version().c_str())
const string SMTClusterPack::version() {return "$Id";}
```

It is very easy to introduce new “hooks” or interfaces into the framework that pass data other then the *edm::Event*. If packages need to communicate objects between themselves without using the class *Event*, then this facility should be considered.

Framework *Package*s can be constructed outside the framework main routine using the framework’s *makePackage( )* function from header *framework/Testing.hpp*. This feature has existed for a long time and is documented in the **Framework User’s Guide**.

5.2 EDM Usage

5.2.1 Dangerous use of *TKey*

Further investigation need to be done to determine why mutable is used to such extents in the tracking chunks. Many uses of the EDM class template *TKey* return any of the chunks of that type in the event, this may not be desirable. We have found this use is common in other code we have reviewed, and would recommend that this ability be removed from the class *TKey*; to make up for this, the efficiency of *TKey::findAll( )* would have to be improved.

5.2.2 Mutable members

In *GTrackChunk* (package **gtr_evt**), all the data members are declared *mutable*, thus completely violating the mechanism by which the EDM assures that objects put into the *Event* aren't modified after insertion. It is very difficult to tell *GTrackChunk*s are logically *const*, which is what is required by the EDM. We have seen mutable members in many of the chunks and had a difficult timing telling if they are cached data, or data that needs to be passed from package to package and modified. Even if the current version of *GTrackChunk* does not violate logical const-ness, this widespread use of mutable makes this class prone to maintenance trouble.

5.3 Magnetic field

The magnetic field class (from **mag_field**) is not used; instead, a constant (*double* or *float*) value is used, presumably to indicate a uniform field in the *z* direction. The value of the constant is provided from a couple of sources:

- read from an *obs* file, *e.g.* cft_init.obs, from **cft_obs**;
- hard-coded into a source file, *e.g. GtrFindImp* in **gtr_find**.

We found no places in which the magnetic field was initialized from a value read from an RCP file.

There are many classes which expect this simple representation of the magnetic field. It is unclear exactly how much work would be involved in making the appropriate classes be aware of a more complex (non-uniform) magnetic field, but it is clearly not a trivial task.

5.4 Use of the geometry system

How is it made certain that the geometrical description of the DØ detector used by the tracking reconstruction is the same as that defined by the DØ geometry system? We haven't yet found where the DØ geometry system is used by the tracking code. This may have been only because of insufficient time for us to investigate.

5.5 DOOM Objects in SMT

Almost all of the data classes in the SMT code are derived from d0_Object. We have not been able to figure out why this is the case.

6 Examples

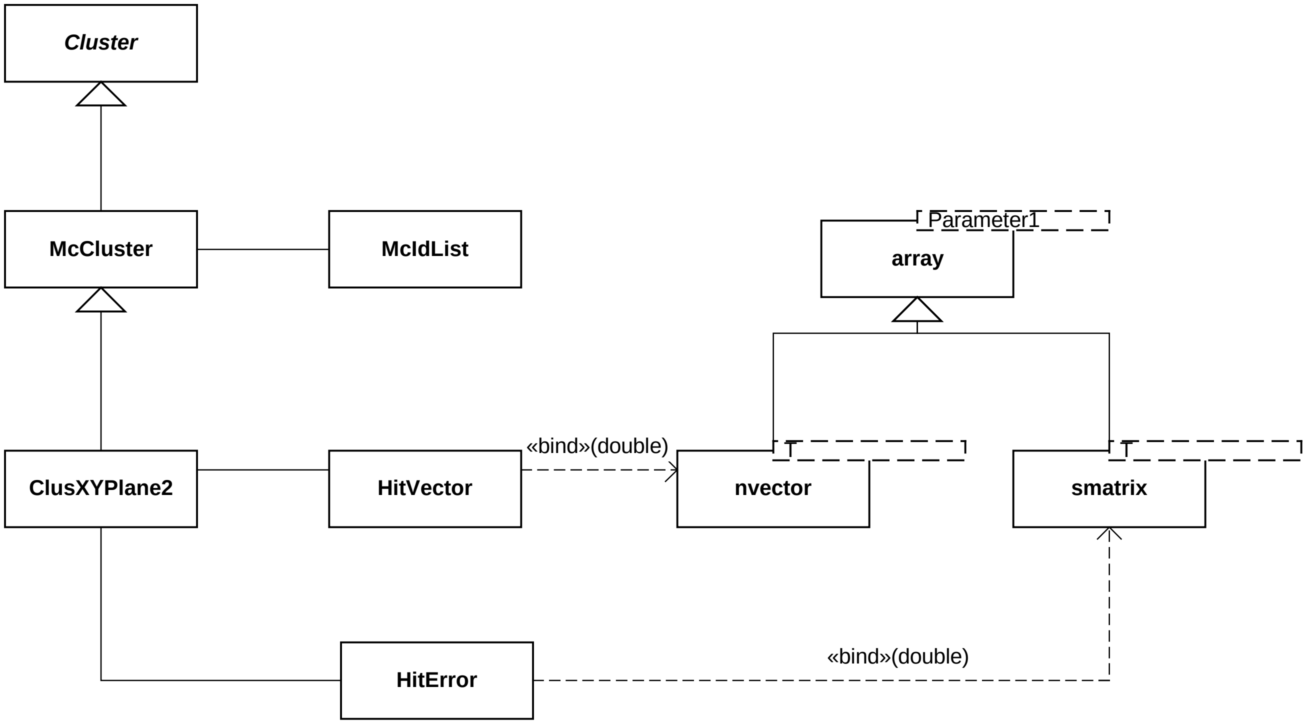

Figure 1 ClusXYPlane2 and related classes.

6.1 *Analysis of* ClusXYPlane2

As an example of unnecessarily large classes, let's consider the example of *ClusXYPlane2*. This class is similar to several other *Cluster* subclasses, and was picked at random to be the subject of detailed investigation. This class, and some of those closely related to it, are diagrammed in Figure 1 (page 11).

The size of a single instance of *ClusXYPlane2* is considerable. The *HitVector* associated with each *ClusXYPlane2* contains two doubles (16 bytes total). Each *array* (from which *HitVector* derives) also has an overhead of one int and one pointer to its data, plus an additional vtable pointer (12 bytes total); this gives a total of 28 bytes for the *HitVector*. Similarly, the *HitError* object contains 3 doubles, to hold a symmetric error matrix, for a size of 24 bytes. Again, since *HitError* derives from *array*, it carries an extra overhead of 12 bytes, for a total of 36 bytes for the *HitError*. Not shown on the diagram (for want of space) is its *SurfXYPlane* data member; this contains two doubles (16 bytes) plus a vtable pointer, for a total of 20 bytes. Finally, *ClusXYPlane2* inherits from *McCluster*, so it also contains a *McIdList* (which is a *vector* of ints); for an empty *vector*, the size is 16 bytes; *ClusXYPlane2* also carries its own vtable pointer, for 4 more bytes. The inheritance of *McCluster* from *Cluster* (and from *TrfObject* and *ObjType*) carries no additional size overhead. This gives the total size of a *ClusXYPlane2* instance of 104 bytes. In addition to its sheer size, it is important to note that this memory is not contiguous — the two doubles for the *HitVector* are held in one region of memory, the three for the *HitError* are held in a different region of memory, and those of the rest of the object are held in a third location. Figure 2 (page 12) shows this memory layout.

We would propose an alternate design for this class. First, we would start with a 2D position class, which would hold the two double measurements by value, and which would need no virtual functions; this would

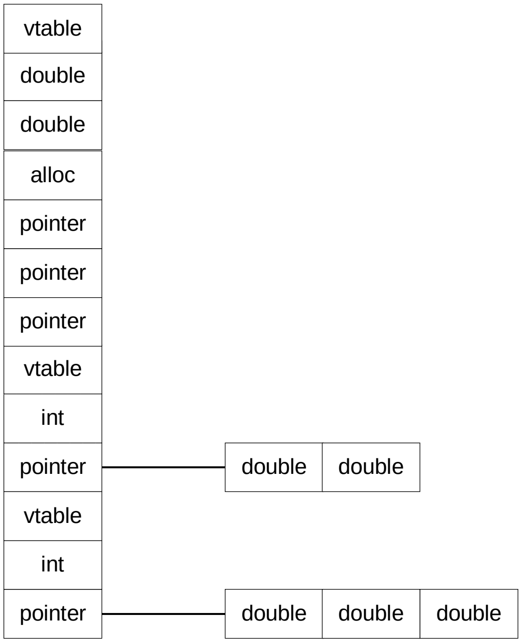

Figure 2: Memory layout of ClusXYPlane2.

replace *HitVector*, and have a size of 16 bytes. Similarly, *HitError* could be replaced by a 2D measurement error class, which would contain 3 doubles, for a size of 24 bytes. Since position measurements may be needed in one, two or three dimensions, a class template — templated on both variable type (float or double) and also on number of measurements — could be used; similarly, a class template could be used to replace *HitError*. With no other modifications, this would reduce the size of a *ClusXYPlane2* object from 104 bytes to 80 bytes. As will be mentioned later, we would also add a reference count data member (probably to the base class *Cluster*), which would keep track of how many things point to a specific *Cluster* instance, making it inherently reference-counted; this would raise the size to 84 bytes. Furthermore, the proposed design would provide better locality of reference, since a single memory area would be used for the entire object.

Next, we consider the manner in which instances of *Cluster* subclasses are more often used. Very often they are held in a *ClusterList*, which is a typedef for *list<ClusterPtr>*. The class *ClusterPtr*, and its related classes, are shown in Figure 3 (page 12). Let us determine the size and the memory layout for a collection (we'll use *vector*, to be concrete) of *ClusterPtr* objects. (*ClusterList* is actually based on *list*, but we observe no case in which the special behavior of *list* is important, and suspect that *vector* would be a superior choice).

A *ClusterPtr* inherits from *ClusterPtrBase*, which is a typedef for *Ptr<Cluster, SharedDeletePolicy>*. A single instance of *ClusterPtr* is composed of a *TYPE** (4 bytes), a *SharedPolicyTable** (4 bytes) and vtable pointer (because *Ptr* has a virtual destructor); this is a total of 12 bytes. The *SharedPolicyTable* contains a bool (4 bytes, because of the alignment requirements on a 32 bit system), and two ints (4 bytes each), for a total of 12 bytes. Thus the first *ClusterPtr* made to point to a given cluster costs 24 bytes; each copy made from the first costs an additional 12 bytes (because the *SharedPolicyTable* is not copied).

So, if we make a *vector* of *ClusterPtr* objects, and fill it with NCLUS pointers, we end with a *vector* requiring:

- 16 bytes (for empty *vector*);
- 24 * NCLUS bytes (for the *ClusterPtrs*);
- 104 * NCLUS bytes (for the clusters objects, if they are all instances of the class *ClusXPlane2*).

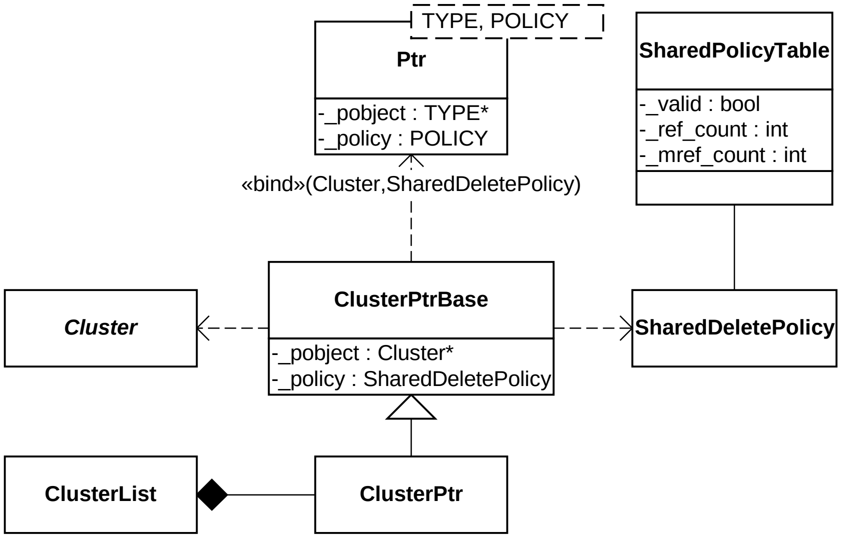

Figure 3: *ClusterPtr* and related classes.

If we have 100 such clusters, the *vector* is 16 + 2400 + 10400 = 12816 bytes.

Our suggested redesign gave an 84 byte cluster (including the reference count, built into the cluster base class). Our *ClusterPtr* only needs to be 4 bytes big, to hold the *Cluster**, and it has to manipulate the reference count in the object — but this doesn't require any virtual functions in the pointer class. So, the size of a *vector<ClusterPtr>* carrying NCLUS pointers is:

- 16 bytes (for the empty *vector*);
- 4 * NCLUS bytes (for our *ClusterPtrs*);
- 84 * NCLUS bytes (for our clusters, assuming they're all *ClusXYPlane2* clusters).

If we have 100 such clusters, memory required for the *vector* is 16 + 400 + 8400 = 8816 bytes. The existing design's *vector* is 1.45 times larger. If we have many *vectors* sharing the same pool of clusters, the comparison is more extreme. The existing *ClusterPtrs* will occupy 24 bytes for the first one pointing to a given cluster. If that particular pointer is copied, each copy costs an additional 12 bytes (because we don't have to pay the 12 bytes more for another *SharedPolicyTable* — we share the one).

Each addition *vector* for the current design (assuming the clusters are already paid for, and the *SharedPolicyTable* is paid for) costs 16 + 12 * NCLUS bytes; for our suggested redesign, it costs 16 + 4*NCLUS bytes. In the limit of large vectors, when the overhead of *vector* itself is negligible, the current design makes vectors that are 3 times larger than our suggested redesign. Since collections of *ClusterPtr* are used widely, the total savings in memory usage may be significant. Perhaps more importantly, the locality of reference for the redesigned classes will be significantly better.

6.2 *Analysis of* ClusFindZPlane2

The following code illustrates a whole series of problems that occur throughout the code. Each of the problems is label with a number and explained below.

```
// relevant method call
HitVector ClusZPlane2::get_hm() const;

// relevant data member type from ClusFindZPlane2
①
typedef std::multimap<double,ClusterPtr,std::less<double> > ClusterMap;

int ClusFindZPlane2::add_cluster(const ClusterPtr& pclu) {
       double keyval; // declare the key value

       // Extract the cluster position depending on type
       ②
       if ( pclu->get_type() == ClusZPlane2::get_static_type() ) {
              ③
              const ClusZPlane2& clu = (const ClusZPlane2&) *pclu;
              assert( clu.get_surface() == _szp );
              ④
              keyval = clu.get_hm()(ClusZPlane2::IX);
       }  else {
              ⑤
              assert(false);
              return 1;
       }

       // Check there is no other cluster at this position.
       ⑥
       ClusterMap::const_iterator iclu = _clusters.find(keyval);
       ⑦
       for ( ;  iclu!=_clusters.end(); ++iclu)
       {
              assert( (*iclu).second != pclu );
       }

       // Store the cluster.
       clusters.insert( ClusterMap::value_type(keyval,pclu) );

       return 0;
}
```

- ① The use of a *multimap* where a double is a key immediately caught our attention. This does not seem to be appropriate because of round-off errors, which could easily lead to keys which are indented to match, but which fail to do so. The particular use of *multimap* in this class is common amongst the *ClusFind* classes. This *multimap* appears to be high overhead for what it is used for — keeping the clusters in order as they are added. A substitute could be a *vector* that is given an initial space reservation. The clusters could be pushed onto the *vector* when added. A new verify phase can be added that will sort the cluster *vector* and check for duplicates (this is a very efficient process).
- ② Using the *ObjType*/*ObjTable* utilities in this fashion appears to be a substitute for *dynamic_cast*. The problem here is that this is a very expensive operation compared with *dynamic_cast*. It appears that a call to *get_type( )* looks like this: *get_type( )* → vtable[x] → *get_static_type( )* → *map<string,funcptr>* lookup. A *dynamic_cast* is typically a lookup in the vtable for a structure, then retrieval of an integer value from that structure. This use of *ObjTable* utilities should be removed.
- ③ Hard casting in this fashion is by no means guaranteed to work, according to the C++ standard. In fact, we have found specific instances where it does not work. The use of *dynamic_cast* would remove this possibility of failure.

- ④ The *get_hm( )* method of the cluster class returns an instance of a *HitVector*. This is very inefficient considering that all that is need is a single number. In fact, the returned *HitVector* is a temporary that will not (and can not) be used again.
- ⑤ This code will abort when not optimized and return a 1 if optimized. Return codes like this do not give any indication as to the severity of the problem; an exception may be more appropriate. A problem with return code is that they are not checked in many places, this appears to be the case here.
- ⑥ The *find* method of *multimap* return any one of the exact matches for this key. This is likely to not be the first one and will lead to undesirable results.
- ⑦ The code inside this loop is only present in the unoptimized executable. This loop will only be wasting time in the optimized executable. This loop will most likely not start at the correct place because of the use of find( ).

7 Conclusion

Because of the size of this system, we were not able to deal with the design at the same level of detail with which we review other software systems. We therefore have no observations on the global structure of the design. Instead, we have concentrated primarily on coding details of the sampling of classes and functions which we were able to analyze in the time available.

In this document, was have given an overview of some of the features of the global tracking software that we would expect to lead to difficulty in maintenance, and also to inefficiency (meaning the speed of the code, not the tracking efficiency). Our discussion of efficiency is limited because of the unavailability of an optimized tracking executable, with which to perform profiling.

When such profiling becomes possible, we expect analysis of this system will be difficult and that there will be no one thing or even a hand full of things that are causing trouble. Performance penalties are likely to be spread across many of the classes, methods, and packages.

During the past few weeks we have attempted to study (by performing small experiments) the effects of memory usage on performance. It is widely publicized and understood that memory usage greatly effects the amount of work the CPU can do and that it can be a significant bottleneck in a system. In addition, it is well known that one reason modern CPUs can perform at such high speeds is because of the memory caches. Therefore, locality of reference is important. The reason that this is relevant here is that heap memory and pointers are used very liberally in this system at all levels. Excessive use of pointers, lookup tables, heap memory, and return of complex data structures by value can cause cache misses and CPU pipeline flushes that can dramatically affect performance.

Our studies do show that object size directly relates to the speed at which it can be manipulated. If the object size is decreased to one half its size, it will take one half the time to move it around. If pointers to data are removed (such as *vectors* or *array*s) and replaced by classes with fixed size (or even with real C-style arrays), that performance is increased. This problem is not easy to analyze and quantify and requires further study.